Strings, I/O Operations, and File Management

16

LEARNING OBJECTIVES

After completing this chapter, you will be able to understand:

- All about strings.
- Various string operations such as concatenation and split.
- StringBuffer and StringBuilder.
- I/O operations.
- InputStream and OutputStream.
- File management.
- Access files and manipulation of files.

16.1 | Introduction

The String class of Java programming language is one that interests many due to its unique characteristics. Unlike the strings of other programming languages such as C++ or C, which are essentially only arrays of chars, strings in Java comprise immutable sequences of Unicode characters that make them unique in many ways.

To make the creation and manipulation of strings easier than it is in most programming languages, the String class in Java offers several different features. Owing to these features, there are a few different techniques that can be used to create and modify strings according to one's requirements or preferences and to ensure that they add value to the application, program, or block of code where they are being used. The creation of strings, for example, can be done using either a string literal, or by using a constructor and calling it for the creation of a String instance. Both techniques will be explained in detail in subsequent sections in this chapter.

16.2 | Role of Strings in Java

Before we discuss the different reasons and ways in which Strings can be used in Java, it is important examine a few facts that every programmer of the Java language should know about strings. We already know that Strings in Java are unlike strings in other languages such as C and C++. They are not simply arrays of chars, and there is a lot that makes the Strings in Java unique.

One main feature that makes Strings in Java important is concatenation. With just a simple "+" sign, you can easily concatenate the contents of two or more strings – something which is not possible with other objects in the Java language, such as Circle or Point. Additionally, since the contents of Strings in Java is as immutable as mentioned above, modifications cannot be made to the contents of any string that is being used in any application, program, or block of code. This means that functions like `toLowerCase()` or `toUpperCase()` will create an entirely new string instead of modifying or changing the contents of the current string, and this new string will then be returned.

The use of null characters for the termination of strings in other languages such as C or C++ is another way in which Strings of Java are different. Strings in Java are objects that are backed by character arrays. If you wish to view the contents of the String in Java in the form of the character array that represents it, you can use the `toCharArray()` method of the String class.

Comparison of Strings in Java is also far easier than it is in other programming languages. Instead of following a lengthy process for the comparison of strings, all you need to do is use the `equals()` method for the comparison of two strings that are being used in any program, application, or block of code. This is possible because the String class of Java overrides the `equals()` method, making comparison of strings extremely easy, convenient, and hassle-free.

Similarly, searching for substrings within a string of Java is also far easier than in other languages. With the help of regular expressions and simple methods such as `indexOf()` and `lastIndexOf()`, parts of strings can be searched for and values

returned if a match is found. Strings can also be trimmed and split into multiple other strings using regular expressions and used separately for a variety of purposes in any application, program, or block of code in Java.

Now that you are aware of the variety of benefits and features that the String class and strings in Java language offer, we will discuss the String class and how strings work in Java programming.

You have learned about how memory is managed by garbage collection in Java language in Chapter 15. Here is a quick recap. There are two major entities that are used for memory management in Java – the heap and the stack. The stack is used for the execution of operations and processes as they are called in the block of code, program, or application. On the other hand, the heap has more to do with storage of contents that are required for the effective running and execution of the code, program, or application.

But what does that have to do with strings? The answer is not exactly simple or straightforward by any means. Strings and the String class of Java programming language are given special treatment, and any string literals that are used in the programming language are assigned a special storage space in the heap memory known as *string constant pool*. Whenever string objects are created using string literals in Java, these objects are stored in the string constant pool. On the other hand, when string objects are created using the new keyword, they are treated just like other objects and are sent to the heap for storage purposes. The following is an example of how string objects are created using string literals:

```
package java11.fundamentals.chapter16;
public class LiteralExample {
    public static void main(String args[]) {
        String message = "Hello World! I love Java";
        System.out.println(message);
    }
}
```

The String Object of the string message shown below with contents "Hello World! I love Java" has been created with the help of a string literal. It will go in the string constant pool instead of being sent to the heap memory like other string objects.

```
Hello World! I love Java
```

Similarly, String Objects can also be created using the new keyword as follows:

```
package java11.fundamentals.chapter16;
public class KeywordExample {

    public static void main(String args[]) {
        char[] javaArray = { 'I', ' ', 'L', 'O', 'V', 'E', ' ','J', 'A', 'V', 'A' };
        String javaString = new String(javaArray);
        System.out.println(javaString);
    }
}
```

The example above shows how a string object can be created using the `new` keyword. See the output below.

```
I LOVE JAVA
```

As mentioned earlier, string objects that are created using this method are treated like normal objects and are sent to the heap, where they are stored along with other objects and variables that are important for the execution of codes or programs.

What most people do not know about the string constant pool is that pool space is allocated to objects depending on the content of the string object in question. This means that when objects are sent to the string constant pool, they are checked to ensure that there are not any two objects that have the same content.

Whenever a new object needs to be created using string literal, the Java virtual machine (JVM) goes through the content of the object that the user wants to create, and then double-checks the content of the objects that are already available in the pool. If an object with content that is the same as the one that needs to be created already exists in the pool, the reference of this object is returned and the new object is not created. The new object will only be created if the content in it is unique and distinct.

However, this is not the case when the new keyword is used for the creation of a new string. If you attempt to create a new string using the new keyword, it will be created whether or not it contains the same contents as an existing string. This shows that two string objects present in the heap memory can have the same contents, but that is not the case with string objects present inside the string constant pool. This can also be proven using the == operator, which only returns as true if the physical address of both objects being compared is the same.

```
package java11.fundamentals.chapter16;
public class StringObjectComparison {
    public static void main(String[] args) {
        // The following strings are being created using literals
        String literal1 = "xyz";
        String literal2 = "xyz";
        System.out.print("Comparison using == operator for literals : ");
        System.out.println(literal1 == literal2); // The output of this line of code will
be true
        // The following two strings are being created using the new operator
        String keyword1 = new String("abc");
        String keyword2 = new String("abc");
        System.out.print("Comparison using == operator for objects : ");
        System.out.println(keyword1 == keyword2); // The output of this line of code will
be false
    }
}
```

The above program creates the following output.

```
Comparison using == operator for literals : true
Comparison using == operator for objects : false
```

Since the two string objects created using string literals having the same contents will also have the same physical address, the output of the first comparison returns as "true". On the other hand, even though the contents of the string objects created using the new keyword are also the same, this second comparison will return as "false" because each of these string objects will have different physical addresses in the heap. This further proves that two objects cannot have the same contents if they need to be stored in the string constant pool, whereas it is completely normal for them to coexist in the heap memory. See Figure 16.1 to understand how String objects are located on Java heap and how they are compared to each other.

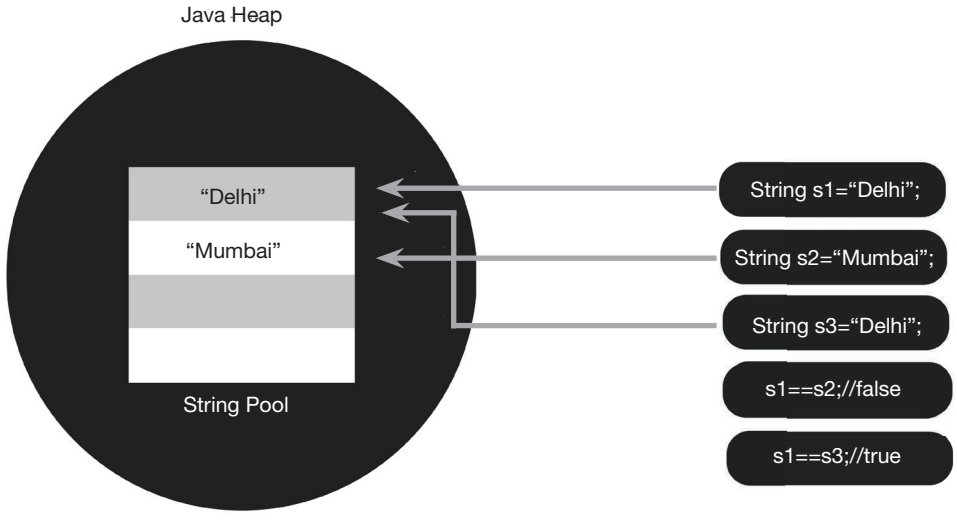

Figure 16.1 String Pool on Java heap and == Comparison.

Understanding the difference between the strings that go on the heap and those that go on the stack is imperative in order to understand which objects can potentially be garbage collected more easily, once they are no longer in use. There are essentially four major types of references that exist in Java:

1. Strong references.
2. Weak references.
3. Soft references
4. Phantom references.

For the sake of explanation, consider the following example:

```
Greeting hello = new Greeting();
```

In the line of code above, "hello" acts as a strong reference to the `Greeting()` object.

Flash Quiz Can the garbage collector collect a String object?

In case an object does not have a strong reference (and only has a soft reference), there is a high possibility of the memory of said object being recollected in case the garbage collector needs additional memory for operations. On the other hand, if an object only has a weak reference assigned to it, the garbage collector will reclaim the memory of this object in the next cleaning phase, regardless of whether or not enough memory already exists.

If an object does not have a strong, weak, or soft reference, then the reference that it has is often called *phantom reference*. Unable to be accessed directly, these references are not known to many programmers or developers of Java, which makes them particularly interesting. Another important fact is that whenever the `get()` method is used on phantom references, they always return as null.

The most popular and powerful types of references – strong references – are used extremely common for programs, blocks of code, and applications that are developed using Java. Objects can be easily created in Java language and assigned references. It is important to note that whenever an object has a strong reference, it can never be garbage collected.

Since strings are given special treatment in Java, the same holds true when it comes to garbage collection of string objects that are used in blocks of code, programs, or applications created using the language. As you already know, every time a thread is created and started, it has its own stack memory. Here, it is important to note that even if an object is present in the heap, but is no longer being referenced by the stack, it becomes eligible for garbage collection. Even if an object in the heap has strong references to other objects present within the heap, they become eligible for garbage collection and will eventually be removed or deleted if they do not have a reference from the stack.

Here are a few facts regarding the garbage collection process and how it really works:

1. Garbage collection is an automatic process in Java. What this means is that starting the process is at the discretion of the JVM.
2. Garbage collection is actually an expensive process. This is because the running of the garbage collector essentially puts all the other threads of the application, program, or block of code on hold, until the garbage collection process is completed.

More complex than simply calling a method and freeing up memory, the garbage collection process essentially entails the use of the mark-and-sweep mechanism that helps JVM decide which objects need to be kept alive for the program, block of code, or application to run and be executed effectively. This helps us understand that even though garbage collection essentially works automatically in Java, certain objects and processes can still be left alive to ensure that the quality, efficiency, and/or performance of the application, program, or block of code in use is not being compromised in any way.

Flash Quiz What are some of the most common string operations in the Java programming language?

16.3 | Types of String Operations

There is a wide variety of operations that can be applied to strings when used in Java language. As mentioned earlier, strings in Java can be concatenated and split, and they can also be formatted. Since these operations are extremely important and commonly used for the creation of codes, we will explain them in detail along with examples of codes to help you understand how exactly each operation can be used in your own applications or programs.

16.3.1 Concatenation

Concatenation is one of the most commonly used operations when it comes to strings in Java language. Concatenation in Java is the name of the process that is used to combine the contents of two or more strings to create a new string. There are two major methods that can be used to concatenate two strings in Java:

1. The first method involves the use of the "+" operator.
2. The second method involves the use of the `concat()` method of the String class.

16.3.1.1 Concatenation Using the Addition "+" Operator

Concatenation using the addition "+" operator is the most commonly used technique to add the contents of two strings in Java. It is important to note that whenever you want to add two or more string literals together, they should be within double quotes.

For instance, if you want to combine the strings "Hello" and "people of the world!", you should write the line of code as follows:

```
"Hello" + " people of the world!" //result: Hello people of the world!
```

Here, it is also important to note that for your sentence to read properly, you should ensure that you properly add spaces in the double quotes.

You can also use the concatenation option for printing outputs in Java. The code to add the contents of the same two strings as mentioned above and print the output is as follows:

`System.out.println("Hello" + " people of the world!"); //the output Hello people of the world!` will be printed on the screen.

1. **Combination of strings on more than one line:** When it comes to string literals, Java does not accommodate the contents of a string to span multiple lines. This is another area where the concatenation option can come in handy.

   With the help of the concatenation operation using the "+" operator, you can create a string literal that spans multiple lines, shown as follows:

```
String PopularQuote = "An eye for an eye" +
" will make the whole world blind."
```

2. **Concatenating variable objects:** While the "+" sign is often used as an arithmetic operator, the rules change considerably in case one of the operands with the "+" sign is a string. In such a case, the other operand is also converted into string form to ensure that it makes sense when concatenated with the operand of String type. Let us take a look at an example:

```
float weight = 50.0;
System.out.println("My weight is " + weight);
```

The output for the line of code written above would be:

```
My weight is 50.0
```

In the example above, `weight` is a float variable, so the "+" operator will first convert the operand to String type and then concatenate the two strings, as it normally would. Even though it is not visible to the end user or programmer who wrote the block of code, the conversion from float type to String type is done by calling the `toString()` method. This shows that concatenation operation does a lot more than just combine the contents of two strings, and that background operations will also be done if needed.

QUICK CHALLENGE Consider two strings s1 and s2. Assign values to each like s1="ABC" and s2="XYZ". Now try using "*" operator on those two and note down the result.

16.3.1.2 Concatenation Using the `concat()` Method

The second technique that can be used for concatenation of two strings in Java is by using the `concat()` method. Whenever the `concat()` method of the String class is used, the method is applied to the first string that needs to be added to form the result, and the second string that needs to be concatenated is taken as a parameter. Let us see the following example:

```
public String concat (String myStr)
```

The line of code above shows how `concat()` method takes the second string as a parameter to add, or concatenate, it with the first string.

The following is another example of how an entire block of code is written to concatenate two strings using the `concat()` method of the String class:

```
package java11.fundamentals.chapter16;
public class StringConcatExample {
    public static void main(String args[]) {
        String myStr = "My Favourite Programming Language";
        myStr = myStr.concat(" is Java");
        System.out.println(myStr);
    }
}
```

The output of the block of code above is shown below.

```
My Favourite Programming Language is Java
```

This shows that the `concat()` method works in a way that is different from the "+" operator. There are quite a number of other differences between these two techniques.

1. While the "+" operator can be used to concatenate objects of variable types, the `concat()` method can only be used to combine objects of the String type. What this means is that the `concat()` method of the String class will only work effectively in case it is called on a variable of String type and has a parameter of the String type that needs to be concatenated.

   This makes the `concat()` method a lot more limited than the "+" operator. The latter is a lot more convenient and hassle-free since it can convert variables of a number of data types into String type efficiently and effectively, allowing this operator to offer a wider range of benefits and usages than the `concat()` method.
2. The second major difference between these two methods is that an exception is thrown by the `concat()` method if the object that is entered as a parameter has a null reference. This means that the `concat()` method of the String class throws a NullPointerException whenever a parameter has a null reference. On the other hand, the "+" operator treats the second operand as a null string and still concatenates it with the first string or operand.
3. Unlike the "+" operator that can be used to concatenate multiple strings, the `concat()` method of the String class can be used for the concatenation of only two strings at a time.

Due to the reasons mentioned above, it goes without saying that the "+" operator is used more commonly for concatenation of strings in Java than the `concat()` method. However, since there are significant differences in the working of both of these techniques, the performance and efficiency of applications will also differ depending on the technique that is being used for concatenation.

16.3.2 Splitting Strings

Another extremely common operation that is performed on strings in Java is splitting. There are several different reasons why you might need to split a string into two or more parts, which is why it is important for you to understand the working behind the `split()` method of the String class.

The `split()` method of the String class in Java splits or divides the input string into multiple parts based on the regular expression that is entered as a parameter. The result of this operation is an array of strings that are divided according to the regular expression that was input as the parameter. In one variant of the `split()` method, you even have the option of entering your desired limit for threshold of the result that will be the output.

For the first variant of the `split()` method, the only thing that is required is the string that needs to be split. All other operations and workings will be done by the Java programming language itself.

The following is an example of how the `split()` method works without a limit:

```
package java11.fundamentals.chapter16;
public class StringSplitExample {
    public static void main(String args[]) {
        String myStr = "My Favourite Programming Language : Java";
        String[] arrOfStr = myStr.split(":");
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The result of the block of code given above as an example is shown below.

```
My Favourite Programming Language
Java
```

This shows that it is not necessary for one to enter any limits when using the `split()` method in the Java. In the block of code mentioned above, the String `myStr` was used as an example, and the string was split by the ":" as mentioned in the following line:

```
String[] arrOfStr = myStr.split(":");
```

When printed, the block of code gave us the desired result.

As mentioned earlier, the other variant of the `split()` method of the String class in Java requires the user to add a limit to the result. This limit is entered as an integer parameter. It is important to note that there are three types of values that the limit integer can take:

1. Positive.
2. Negative.
3. Zero.

Since the value that you select for the limit has an effect on the result that you will get after the string is split, it is important to understand how the value of the result can potentially change according to the type of value that you select for the limit.

1. **Positive:** In case the user selects a positive limit for the result, the pattern will be repeated a maximum of limit –1 times. Additionally, the length of the final array will not be more than the size of the string itself, and the last entry of the final array will contain the remaining part of the string after the pattern was last matched.
2. **Negative:** In case the value for the limit entered by the user is negative, the pattern will be repeated as many times as possible. When the value for the limit is set as a negative integer, there will also be no limits on the size of the final resulting array.
3. **Zero:** When the value of the limit variable is set as zero, the pattern will be repeated as many times as possible. Again, the final resultant array can be of any size. It is, however, important to note that empty strings will be discarded from the final result.

QUICK CHALLENGE

Write an algorithm which can split any string without using the `split()` method. Print timestamp before and after the code to verify which method is faster.

The following is an example to help you understand how the limit works for the `split()` method:

```
package java11.fundamentals.chapter16;
public class StringSplitWithLimitExample {
    public static void main(String args[]) {
        String myStr = "I@love@java";
        String[] arrOfStr = myStr.split("@", 2);
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The output of the block of code written above will be as follows.

```
I
love@java
```

Since the limit was set as 2 and the string was to split at "@", there are only two substrings in the result, which is seen in the output.

Similarly, there can also be multiple different characters that can be entered as the parameter at which the string will be split. Each of these characters will have to be explicitly mentioned when they are entered as parameters. The following is an example of how this works:

```
package java11.fundamentals.chapter16;
public class StringSplitOnMultipleCharactersExample {
    public static void main(String args[]) {
        String myStr = "My, Favourite @Programming?Language.Java";
        String[] arrOfStr = myStr.split("[, ?.@]+");
        for (String piece : arrOfStr) {
            System.out.println(piece);
        }
    }
}
```

The result of the block of code written above will be as follows.

```
My
Favourite
Programming
Language
Java
```

Since no limits were set for the maximum number of results, the program showed 5 different results, each of which were separated by one of the characters mentioned in the regular expression.

All of the examples mentioned above show that by playing around with your regular expressions and the limits that you set for the number of results that you need, you can modify the types of results that you will get. Moreover, you can also decide the number of substrings that your input string will be divided into.

As mentioned at the beginning of the chapter, strings in Java language are immutable, which means that their contents cannot be changed or modified. This is the reason why new strings have to be created whenever an operation needs to be performed on any string. Fortunately for programmers and application developers of Java language, mutable strings have also been accommodated. Different options are now available for the manipulation of strings without burdening the machine too much, or producing excessive amounts of garbage.

Thanks to the StringBuilder and StringBuffer classes, manipulation of string objects in Java is far easier than it would be if only the String class existed. The StringBuffer and StringBuilder classes allow strings to easily be manipulated without the need for additional strings to be created. This way, you not only save on garbage that can affect the efficiency of your program, but the performance of your application, program, or block of code also gets much better than it was.

Since both StringBuilder and StringBuffer are mutable objects in Java, they offer multiple different manipulation options for the strings that are created. Some of the methods offered by these classes include the `insert()`, `delete ()`, and `append()` methods that are commonly used for the manipulation of strings. While both the StringBuffer and StringBuilder classes are essentially used for the manipulation of strings that are created in Java, there are certain significant differences between the two. These will be discussed in detail in following section.

16.4 | StringBuilder and StringBuffer Explained

The primary reason why both StringBuilder and StringBuffer are used is for the manipulation of strings in Java language, which makes both of these classes mutable. These are unlike the more popular String class, which is used for the creation of strings in Java language. However, this is perhaps the only thing that both the StringBuilder and StringBuffer classes have in common.

Unlike the StringBuilder class, the StringBuffer class is thread safe. This means that this class accommodates and modifies data structures only in a way in which they are guaranteed to be executed safely by multiple threads simultaneously. Since the Java language supports the idea of concurrency, this feature of the StringBuffer class is of particular interest to developers and programmers who try to incorporate concurrency in their programs and applications. The StringBuffer class also contains the `insert()` and `append()` methods, which are popular among programmers who want to manipulate strings in multi-thread environments. Since a wide variety of string operations in Java occur in single-thread environments, the StringBuilder class was created without the thread safety option.

The fact that the StringBuilder class does not support concurrency or work in multi-threading environments is also the major reason why the StringBuilder class is considerably faster than the StringBuffer class. Also, the "+" operator that is used for concatenation of strings in Java also uses either the StringBuilder class or the StringBuffer class internally to perform efficient addition or combination of two or more strings in any application, program, or block of code. Moreover, since StringBuilder was introduced in a later version of Java, most of the problems and shortcomings of the StringBuffer class have been overcome in the StringBuilder class.

QUICK CHALLENGE Give a scenario where you would consider StringBuilder over StringBuffer.

Additionally, since the StringBuilder class does not support the idea of synchronization, its performance, efficiency, and speed are considerably different compared to the StringBuffer class. Synchronization does not only affect processing power negatively, but also accounts for additional overhead that is completely useless.

To validate the claims made in this section that compares the StringBuffer and StringBuilder classes, let us see the example of a program that repeatedly performs the `insert()` and `append()` methods on objects of both of these classes. With the help of this program and different test values, we will show how the performance of both of the classes differs and to what extent.

```
package java11.fundamentals.chapter16;
import java.util.GregorianCalendar;
public class StringBufferVsStringBuilderExample {
    public static void main(String[] args) {
        System.gc();
        StringBuffer myStrBuff = new StringBuffer();
        StringBuilder myStrBuild = new StringBuilder();
        runStringBuilder(myStrBuild);
        // Request Garbage Collection to clear the memory
        System.gc();
        runStringBuffer(myStrBuff);
    }

    private static void runStringBuilder(StringBuilder myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000; j++) {
            myStr.append(":" + j);
            myStr.insert(j, "Hello");
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Builder Append Insert:" + (finish -
begin));
        System.out.println("Memory used String Builder Append Insert:" + (initiateMemory -
stopMemory));
    }

    private static void runStringBuffer(StringBuffer myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000; j++) {
            myStr.append(":" + j);
            myStr.insert(j, "Hello");
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Buffer Append Insert:" + (finish -
begin));
        System.out.println("Memory used String Buffer Append Insert:" + (initiateMemory -
stopMemory));
    }
}
```

The above program produces the following result.

```
Time Taken for String Builder Append Insert:333
Memory used String Builder Append Insert:1603872
Time Taken for String Buffer Append Insert:318
Memory used String Buffer Append Insert:1175040
```

As mentioned above, the value of all variables was changed multiple times, and the program was repeated with each modification to check how the results varied. The value of *j* was changed from 1000 to 50,000, and the program was repeated multiple times for both StringBuilder and StringBuffer. However, there was not too great a difference in the performance of both these classes.

Since the `insert()` method requires a lot of memory and produces plenty of garbage, the same test can also be conducted on both the StringBuilder and StringBuffer classes without this method to get a better understanding of how the efficiency and performance of both these classes differ.

Additionally, since larger numbers of repetitions will give users and readers a better understanding of how much the results differ. The following is another code which is repeated 50,000,000 times to put things into perspective.

```
package java11.fundamentals.chapter16;
import java.util.GregorianCalendar;
public class StringBufferVsStringBuilderWithoutInsertExample {
    public static void main(String[] args) {
        System.gc();

        StringBuffer myStrBuff = new StringBuffer();
        StringBuilder myStrBuild = new StringBuilder();

        runStringBuilder(myStrBuild);

        // Request Garbage Collection to clear the memory
        System.gc();
        runStringBuffer(myStrBuff);
    }

    private static void runStringBuilder(StringBuilder myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000000; j++) {
            myStr.append(":" + j);
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Builder Append:" + (finish - begin));
        System.out.println("Memory used String Builder Append:" + (initiateMemory -
stopMemory));
    }

    private static void runStringBuffer(StringBuffer myStr) {
        long begin = new GregorianCalendar().getTimeInMillis();
        long initiateMemory = Runtime.getRuntime().freeMemory();
        for (int j = 0; j < 50000000; j++) {
            myStr.append(":" + j);
        }
        long finish = new GregorianCalendar().getTimeInMillis();
        long stopMemory = Runtime.getRuntime().freeMemory();
        System.out.println("Time Taken for String Buffer Append:" + (finish - begin));
        System.out.println("Memory used String Buffer Append:" + (initiateMemory -
stopMemory));
    }
}
```

The above program produces the following result.

```
Time Taken for String Builder Append:2200
Memory used String Builder Append:-533019072
Time Taken for String Buffer Append:2401
Memory used String Buffer Append:-14155776
```

When the test was repeated without the `insert()` method as mentioned earlier, a considerable difference was seen in the amount of time that was taken for the execution of the program to be completed by both the StringBuilder and StringBuffer classes. With the help of this updated example, it is evident that the StringBuilder class can perform better than the StringBuffer class even when multi-threading is not used.

Now that you know how string operations work in Java and how the String, StringBuilder, and StringBuffer classes differ from each other, we look into other significant areas of the Java programming language including file operations and I/O operations.

In the sections that follow, we will talk extensively about Java File Class, and how methods and operations can be performed for optimal results and maximum efficiency.

16.5 | Java I/O

Ever wondered why you have to write "import java.io" at the beginning of most, if not all, of your Java codes? We will answer this question by first understanding Java I/O. As the name suggests, Java I/O is what is used at the back end to process all input and output operations to make your programs work seamlessly and efficiently.

All this is done with the help of streams! Streams are used in Java language to not only expedite the input and output process, but to also make all operations and processing seamless in order to provide optimal results. Additionally, the java.io package comprises multiple classes, each of which can prove to be beneficial for a number of different reasons, helping you complete all operations efficiently and effectively. The console for Java comes with three built-in byte streams to make processing easier. These streams are:

1. **System.out:** This is the standard output stream that is used for operations in Java.
2. **System.in:** This is the standard input stream that is used for operations in Java.
3. **System.err:** This is the standard error stream that is used for input output operations in Java.

The following are a few sample codes to show you how each of these streams work to make the input and output process in Java more efficient:

```
package java11.fundamentals.chapter16;
import java.util.Scanner;
public class InputOutputProcessExample {
    public static void main(String args[]) {
        // Following code will create scannerObj object of Scanner class
        Scanner scannerObj = new Scanner(System.in);
        System.out.println("Enter the name of the student");
         // Below line of code ensures that data will be input as string by default
        String studentNAME = scannerObj.next();
        System.out.println("Enter the roll number of the student");
        // Below line of code ensures that data will be input as int by default
        int studentRollNumber = scannerObj.nextInt();
        System.out.println("Enter the marks that the student obtained");
        // Below line of code ensures that data will be input as float by default
        float studentMarks = scannerObj.nextFloat();
        System.out.println("-------Student Report Card-------");
        System.out.println("Student Name:" + studentNAME);
        System.out.println("Student Roll No.:" + studentRollNumber);
        System.out.println("Student Marks:" + studentMarks);
        // Following code is needed to avoid resource leak
        scannerObj.close();
    }
}
```

The above program produces the following result.

```
Enter the name of the student
Mayur
Enter the roll number of the student
17
Enter the marks that the student obtained
100
-------Student Report Card-------
Student Name:Mayur
Student Roll No.:17
Student Marks:100.0
```

After each line of code where an input is requested from the user, the user will be allowed to enter a value according to the data type that is set as default.

To help put things in perspective, here is a line of code that explains how the System.err stream can be used:

```
System.err.println("This is an error message");
```

Now that you understand how the System.in, System.out, and System.err streams can be used, we will discuss the OutputStream and InputStream classes, their most popular methods, and how they differ. See Figure 16.2 to understand the role of InputStream and OutputStream in a Java application.

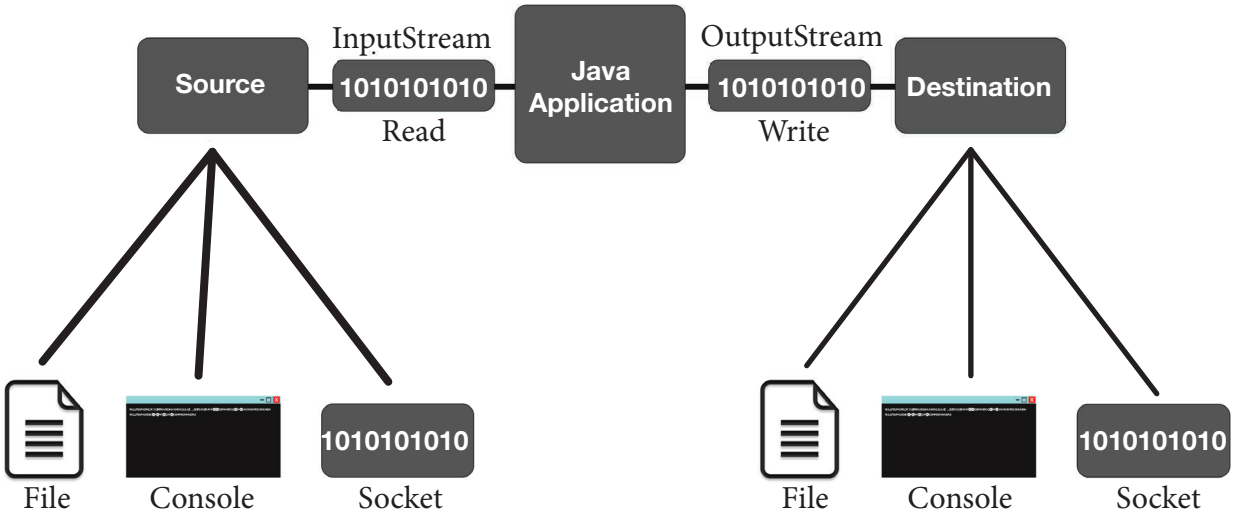

Figure 16.2 Role of InputStream and OutputStream.

16.5.1 InputStream

One of the major classes of the java.io package is the InputStream class. It provides users with a mechanism to help data be input into Java programs and be read without any problems. An abstract super class by definition, the InputStream class provides programmers and developers with the right tools to not only read bytes of data in singular and array form, but to do so selectively with data streams, mark locations within data streams, determine the number of bytes that are available to be read, and reset the current position within a stream of data.

Here, it is important to note that input streams in Java are opened as soon as they are created. The benefit with this feature is that you do not need to explicitly call the input stream whenever it is needed, and the console will take care of that bit automatically. When the input stream is no longer needed and all information that was requested from the user has already been entered, the `close()` method can be used to close the stream explicitly, or wait for the garbage collection process to be completed, which will automatically close and remove the input stream once it is no longer in use and no longer being referenced.

16.5.2 OutputStream

This abstract super class of Java language is essentially used to deal with all outputs in an efficient and effective manner. In addition to providing users with the mechanism and tools to write bytes of data and arrays of bytes, the OutputStream class acts as an interface that is used by multiple other classes to make processing of data more convenient and hassle-free. Much like input streams, output streams can also be closed explicitly by using the `close()` method, or by garbage collection.

Now that you know how both classes differ from each other, we will discuss some of the most popular methods of each of these classes to help you get a better understanding of how they can be used.

16.5.3 Methods of OutputStream

There are several different methods of the OutputStream class, each of which serves a unique purpose and offers a unique set of benefits. We will discuss a few of these methods and explain how they differ from each other.

1. The public void `write(int)` method is the method of the OutputStream class that is used to write a single byte to the output stream that is currently in use. It should be noted that this method cannot be used for arrays of bytes.
2. The public void `write(int[])` method is used for writing an array of bytes to the output stream that is currently in use. The difference between the public void `write(int)` and public void `write(int[])` is also evident from the parameters that each of these methods allow. Since the second method has int[] or integer arrays as a parameter, it goes without saying that this method should be used to write an array of bytes to the output stream that is currently in use.
3. The third method of the OutputStream class is the `flush()` method. As the name suggests, this method flushes the output stream that is being used at any given point of time.
4. The last method, `close()`, is used to close the output stream that is currently being used, removing all references and making it eligible for garbage collection.

16.5.4 Methods of InputStream

The following methods are part of the InputStream class and can be used to input data to be used by the program or application.

1. The first method is the public abstract int `read()` method. It is used to read the next available byte of data in the input stream. Whenever the method reaches the end of a file or stream, it returns –1 as an indication that there is no longer anything that can be read.
2. The second method is the public int `available()` method. It shows the user an approximate number of bytes that can still be read from the current input stream.
3. The third method is the public void `close()` method of the InputStream class works in the same way as the method of the same name in the OutputStream class, and is used to close the current input stream and make it eligible for garbage collection.

What permissions do you need on a server to use the File Management functionality?

16.6 | File Management in Java

File handling or file management is another area of Java that tends to interest a number of people, particularly due to the variety of operations that can be performed on files in the language. We will share some insights into file management in Java programming language.

The FileReader and FileWriter classes are of immense importance when it comes to file management in Java language. Inheriting the OutputStream class, the FileWriter class is used for the creation of files by writing characters. While certain assumptions are made by the constructors of this class, programmers and developers of Java are free to specify values by themselves by creating their own constructors. Some of the constructors of the FileWriter class are as follows:

1. **`FileWriter(File,File):`** As the name suggests, this constructor creates a FileWriter object when a File object is input as a parameter.
2. **`FileWriter(String filename):`** Since the parameter for this constructor is a String, the constructor will create a FileWriter object whenever a file name is input as a parameter.

Here are some of the methods that are available through the FileWriter class:

1. `public void write (int c):` This method is used to write a single character into the stream that is being created.
2. `public void write( char[] stir):` The character array that needs to be inserted to the output stream will be input into this method as a parameter.

Inheriting the InputStreamReader class, the FileReader class is used to read data from any file – one character at a time. It is important to note that the FileReader class can only be used to read data in the form of characters, while the FileInputStream class is used to read data in the form of raw bytes. Here are some of the constructors of the FileReader class:

1. `FileReader(File,File):` As evident from the parameters of this constructor, a FileReader object is created when a File object is inserted as a parameter.
2. `FileReader(String filename):` This constructor creates a FileReader object given that the name of the file that needs to be read from is input as a parameter.

Here are some of the methods that are available through the FileReader class:

1. `public int read():` This method is used to read a single character from the stream that is available.
2. `public int read(char[] cbuff):` This method is used to read characters into an array.

Can you read a file from a remote server?

Summary

With this, we have completed the chapter on Strings and how they work in Java language. We have also described in detail about input and output in Java programming language, and the methods and operations that are available to make processing more efficient and optimized.

In this chapter, we have learned the following concepts:

1. String operations and its usage.
2. Various string functions such as concatenation and split.
3. Differences between StringBuffer and StringBuilder.
4. Various I/O operations.
5. File management with file reader and file writer.

In Chapter 17, we will learn about data structures and its types, such as primitive and non-primitive data structures, and how to use them in a program.

Multiple-Choice Questions

1. Which of the following sentences is false about String in Java?
   - **(a)** We can extend String class like StringBuffer does it.
   - **(b)** String class is defined in java.util package.
   - **(c)** String is immutable in Java.
   - **(d)** String is thread-safe in Java.
2. Which of the following methods of String class can be utilized for testing strings for equality?
   - **(a)** `isequal()`
   - **(b)** `isequals()`
   - **(c)** `equal()`
   - **(d)** `equals()`
3. _____ class is used for reading characters in a file.
   - **(a)** FileWriter
   - **(b)** FileReader
   - **(c)** FileInputStream
   - **(d)** InputStreamReader
4. _____ method is used for reading characters in a file.
   - **(a)** `read()`
   - **(b)** `scan()`
   - **(c)** `get()`
   - **(d)** `readFileInput()`

5. _____ method is used for writing into a file.
   (a) `putfile()`
   (b) `write()`
   (c) `writefile()`
   (d) `put()`

Review Questions

1. What are various operations you can perform on String?
2. Is String immutable?
3. How to read and write content into a file?
4. How do we use StringBuilder?
5. What are the advantages of using StringBuilder over StringBufffer?
6. How do we use InputStream? Give one example.
7. How do we use OutputStream? Give one example.

Exercises

1. Write a program to test the difference between StringBuffer and StringBuilder.
2. Write a program to test all the string manipulations using various functions.
3. Create a program which writes content into a file and read from the file.

Project Idea

Create a job application portal that can accept an applicant's CV and process its data. The program should be able to read the CV file and collect the applicant's name, address, education, employment details and show it on a page.

Recommended Readings

1. Oracle Tutorials: Regular Expressions – https://www.w3schools.com/java/java_strings.asp
2. W3Schools – https://docs.oracle.com/javase/tutorial/essential/regex/
3. Oracle Tutorials: Manipulating Characters in a String – https://docs.oracle.com/javase/tutorial/java/data/manipstrings.html
4. Oracle Tutorials: Basic I/O – https://docs.oracle.com/javase/tutorial/essential/io/